Proceedings of the
Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

The Ondemand Governor

Past, Present, and Future

Venkatesh Pallipadi   Alexey Starikovskiy

Intel Open Source Technology Center

venkatesh.pallipadi@intel.com
alexey.y.starikovskiy@intel.com

Abstract

ondemand is a dynamic in-kernel cpufreq governor that can change CPU frequency depending on CPU utilization. It was first introduced in the linux-2.6.9 kernel. Its simplistic policy provided significant benefits to laptops, desktops, and servers alike by making use of fast frequency-switching features of the processors to effectively power-manage them.

This paper starts with a description of the ondemand governor present in the 2.6.9 kernel: the algorithm and tuning parameters in that governor. In particular, it highlights the significant difference between the ondemand governor vs. the user-level cpufreq governors. This section also includes a brief overview of how to configure and run the ondemand governor.

Next is a discussion of various optimizations to the original ondemand algorithm. Some of these changes were driven by the new processor support of dynamic changing of frequency in multi-core and multiprocessor system environments. This section highlights the challenges of changing frequency in a multiprocessor system environment such as preventing frequency change in one processor affecting other processors. It also discusses relative power/performance data with the ondemand governor and its various optimizations.

This paper concludes with a few ideas about where the ondemand governor is headed in the future, including additional features that are nice to have and how the ondemand governor can be made more useful in a wide range of systems—from handhelds to servers. This discussion touches upon changes that may be required in kernel subsystems other than cpufreq, in order to improve effectiveness of the ondemand governor.

1 Introduction

Most of the latest microprocessors have mechanisms to save power by changing the core voltage and frequency at run time. Section 2.1 gives a detailed view on this.

This technique was first widely deployed on mobile systems due to their battery life requirements, but is now common on desktops and servers as well.

With this technique being widely used in different kinds of systems, there has been a constant stream of optimizations happening in the cpufreq infrastructure itself, in the ondemand governor, and also in low-level drivers and ACPI [4]. Recent changes in cpufreq include the ability to handle CPU hot-plug cleanly and also the ability to deal with processor groups sharing one frequency. This latter feature support is important in multi-core and multi-thread environments, where platforms may have restrictions of running different logical processors at same frequency. This paper will deal in detail with changes in the ondemand governor and low-level governors that use ACPI to identify that all CPUs share the same frequency.

Section 2 of this paper includes a primer on cpufreq. Section 3 covers the motivation for the ondemand governor and the original ondemand algorithm. In Section 4, we discuss various optimizations to the ondemand governor since its original inception into the Linux kernel, followed by some ongoing investigations. In Section 5 we deal with how Frequency Voltage changes happen in presence of multiple logical processors in a package. This issue, although orthogonal to the ondemand governor in itself, is critical for saving power in a multi-core and multi-thread CPU world. Section 6 includes measurement results from our lab, made with various governors and optimizations. We conclude the paper with a glimpse of changes that are likely to come to ondemand in the future. Unless otherwise mentioned, all the kernel-specific details in the paper will be for the 2.6.16 kernel.

2 Background

2.1 Physics

Current CMOS electronics consumes power in three big areas:

$$P = P_1 + P_2 + P_3$$

leakage – current going either through substrate (under schematics) or not fully closed transistors (through schematics) and depends on voltage, thus this part of the power will be proportional to the square of the voltage:

$$P_1 = I_L * U_C = U_C^2 / R_L$$

recharging parasitic capacitance of wires and inputs—depends on both frequency and voltage, linear on frequency and square on voltage.

$$P_2 = U_C^2 / R_P = U_C^2 * C_P * F$$

shoot-through current – happens during the switch of the CMOS circuit then one transistor is already open while opposite to it is just started to close, and thus, is linear proportional to frequency and square proportional to voltage.

$$P_3 = U_C^2 * F / R_S$$

Summarizing the above, consumed power is proportional to square of core voltage and either constant or linear to frequency, depending on which power consumer on a chip dominates. Maximum frequency of the CMOS circuit depends on core voltage as well, and thus, to save power we need to decrease the core voltage, but beforehand set frequency to value, allowed at this reduced voltage. Changing the frequency alone does not bring any significant benefits. This is why drivers that modulate the clock without changing the voltage are ineffective at saving power.

2.2 Cpufreq and governors

cpufreq is the subsystem of the Linux kernel that allows frequency to be explicitly set on processors [3]. cpufreq provides a modularized set of interfaces to manage the CPU frequency changes. Figure 1 depicts the high-level cpufreq infrastructure.

The primary components of this infrastructure are as follows:

Cpufreq module provides a common interface to the various low-level, CPU-specific frequency control technologies and high-level CPU frequency controlling policies. cpufreq decouples the CPU frequency controlling mechanisms and policies and helps in independent development of the two. It also provides some standard interfaces to the user, with which the user can choose the policy governor and set parameters for that particular policy governor.

CPU-specific drivers implement different CPU frequency changing technologies, such as Intel® SpeedStep® Technology, Enhanced Intel® SpeedStep® Technology [6], AMD PowerNow!™, and Intel Pentium® 4 processor clock modulation. On a given platform, one or more frequency modulation technologies can be supported, and a proper driver must be loaded for the platform to perform efficient frequency changes. The cpufreq infrastructure allows use of one CPU-specific driver per platform. Some of these low-level drivers also depend on ACPI methods to get information from the BIOS about the CPU and frequencies it can support.

In-kernel governors. The cpufreq infrastructure allows for frequency-changing policy governors, which can change the CPU frequency based on different criteria, such as CPU usage. The cpufreq infrastructure can show available governors on the system and allows the user to select a governor to manage the frequency of each independent CPU.

Kernel 2.6.16 comes bundled with five different governors. Three of these governors can be run on any kind of CPU that has a low-level driver to change the frequency at run time and can be chosen as default governor at compile time:

Performance governor keeps the CPU at the highest possible frequency within a user-specified range.

Powersave governor keeps the CPU at the lowest possible frequency within a user-specified range.

Userspace governor exports the available frequency information to the user level (through the sysfs) and permits user-space control of the CPU frequency. All user-space dynamic CPU frequency governors use this governor as their proxy.

There are two relatively new governors, ondemand and conservative, capable of frequent load monitoring on CPUs which can do fast frequency switching.

ondemand governor was introduced into Linux kernel in 2.6.9 and rest of this paper covers in detail the algorithm, usage, and recent, ongoing, and future changes to this governor.

conservative governor is a fork of the ondemand governor with a slightly different algorithm to decide on the target frequency. Most of the configuration details of ondemand in this paper also holds true for the conservative governor.

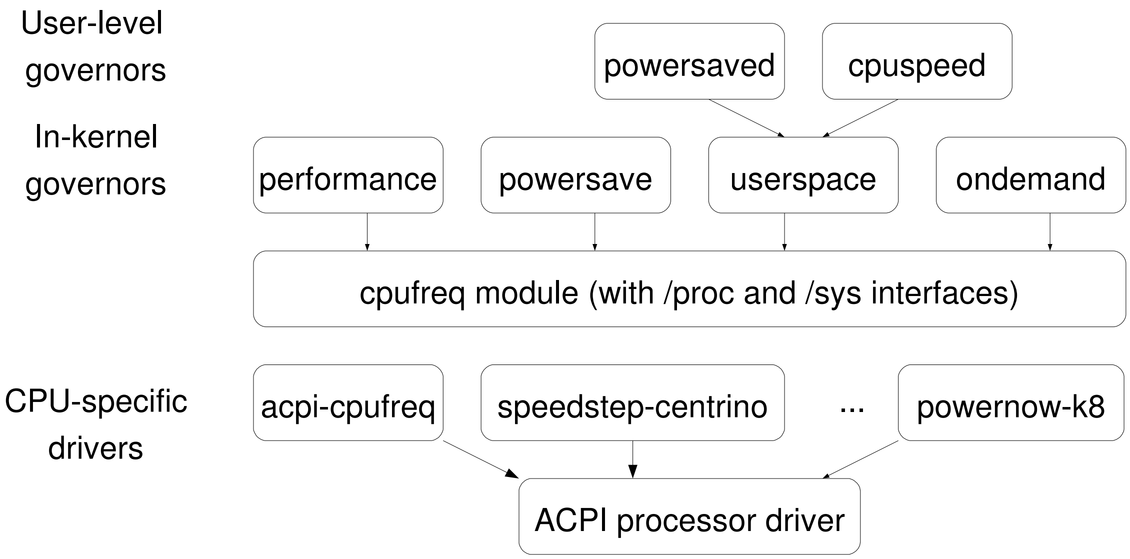

Figure 1: `cpufreq` infrastructure

2.3 `cpufreq` and `sysfs` interfaces

The user interface to `cpufreq` is through `sysfs`. `cpufreq` provides the flexibility to manage CPUs at a per-processor level (as long as hardware agrees to manage CPUs at that level). The interface for each CPU will be under `sysfs`, typically at `/sys/devices/system/cpu/cpuX/cpufreq`, where `X` ranges from `0` through `N-1`, with `N` being total number of logical CPUs in the system.

The basic interfaces provided by `cpufreq` are:

```
linux:> cd /sys/devices/system/cpu
linux:> cd cpu0/cpufreq
linux:> ls -1 -F
affected_cpus
cpuinfo_cur_freq
cpuinfo_max_freq
cpuinfo_min_freq
scaling_available_frequencies
scaling_available_governors
scaling_cur_freq
scaling_driver
scaling_governor
scaling_max_freq
scaling_min_freq
stats/
```

All these files can be read by doing a `cat` and all the writable files can be written to using a `echo` and redirection into the file. `stats/` is a directory and will be discussed in Section 2.4. All the frequency values are in kHz.

Reading `cpuinfo_max_freq` and `cpuinfo_min_freq` will give the maximum and minimum frequency supported by the CPU and `cpuinfo_cur_freq` will read the current frequency from hardware and display it.

`scaling_available_frequencies` lists out all the available frequencies for the CPU.

`scaling_available_governors` lists out all the governors supported by the kernel. Note that the governor modules must be loaded through modprobe for it to appear here. The administrator can `echo` a particular available governor into `scaling_governor` in order to change the governor on a particular CPU.

`scaling_cur_freq` will return the cached value of the current frequency from the cpufreq subsystem. scaling_max_freq and scaling_min_freq are user controlled upper and lower limits, within which the governor will operate at any time.

scaling_driver names the low-level CPU-specific driver that is used to change the CPU frequency.

In addition to above interfaces, the running governor may add some more interfaces of its own, which can be used to manage the frequency or fine-tune the governor.

2.4 cpufreq-stats

The interfaces under the stats/ directory provide the statistics about the usage of frequency changes on any particular CPU. The exact details of the interfaces and their meaning can be found in [1].

2.5 cpufreq-based tools

Reading and changing different fields in the specific sysfs directory by hand on a system with a lot of CPUs can be painful and time consuming. Dominik Brodowski has led the development of cpufrequtils containing a set of tools to make use of cpufreq easier [2].

3 Original on-demand governor

3.1 Motivation

Of the three governors that were there in the kernel before ondemand, the performance and powersave governors were static governors. The userspace governor gave the user (superuser or root) the control to set the frequency on a particular platform. This userspace interface could then be used by the daemons running in userland to manage the CPU frequency over time, depending on the load. There are multiple userspace programs, like cpuspeed and powersaved that can use userspace governor interface and change the frequency based on load. The userspace governors would typically sample the utilization every few seconds, and then take a decision on what frequency to go to for the next sample interval. This method of changing the frequency operates properly with almost any frequency/voltage-changing hardware.

However, hardware capable of low-latency frequency switching can take advantage of software that does more aggressive sampling of utilization and change the frequency more often to suit the workload. For example, Enhanced Intel Speedstep Technology can switch the frequency with latency as low as 10μS. This faster sampling will also help in quick response time for changing workloads, which is critical in servers and will also bring visible benefit for laptop users. Think of CPU frequency going to the max within a few milliseconds after clicking on OpenOffice and compare against clicking on OpenOffice, with the CPU running at low frequency for several seconds, and then increasing the frequency to the maximum.

But doing this frequent polling from userspace may add more overhead due to kernel to user transition and reading/writing /proc/ and /sys files, etc. This was the original motivation behind the ondemand governor. Doing the dynamic frequency change inside the kernel, more often, with less overhead. Also, the kernel is the right place to take the frequency decision as it has lot of other information about the system overall and the particular CPU [7].

3.2 Algorithm

The design goal with the original ondemand governor was to keep the performance loss due to reduced frequency to minimum and to keep the code simple. With that we came up with a simplistic algorithm to dynamically manage the frequencies of different CPUs on the system. ondemand managed each CPU individually, hence on an SMP server, with only one active thread, CPU running the active thread will run at full speed, while other threads will conserve power by running at a lower frequency.

Figure 2 shows the original ondemand algorithm at a high-level.

```
for every CPU in the system
  every X milliseconds
    get utilization since last check
    if (utilization > UP_THRESHOLD)
      increase frequency to MAX

  every Y milliseconds
    get utilization since last check
    if (utilization < DOWN_THRESHOLD)
      decrease frequncy by 20%
```

Figure 2: Original ondemand algorithm

Note that the sampling frequency is a function of transition latency by the hardware and the HZ, as HZ is the unit of idle measurement in the current kernel.

3.3 Configuring ondemand governor

The default governor that the system uses depends on the kernel configuration and the init scripts in the installation. You can check the current governor that is being used on your system by looking at /sys/devices/system/cpu/cpuX/cpufreq/scaling_governor.

If your system is not already using the ondemand governor, you can switch the governor using the cpufreq sysfs interface. To use the ondemand governor, make sure the ondemand governor is configured in the kernel. If it is configured as a module, do a modprobe of cpufreq_ondemand. Then you can change the governor by a simple echo ondemand > /sys/devices/system/cpu/cpuX/cpufreq for each CPU X. Note that in order to do this on every boot, you will have to change/add an init script.

Also note that if your CPU is not capable of fast switching of CPU frequency, then the above echo command may fail and you may continue to use the governor that was set before.

3.4 Tunable Parameters

A single policy governor cannot satisfy all of the needs of applications in various usage scenarios. The ondemand governor exports some tuning parameters to userspace that can fine-tune the algorithm for specific usage scenarios. Below is the list of tunables as they appear in /sys. Note that the will only appear if the ondemand governor is active on this CPU.

```
linux: # cd \
/sys/devices/system/cpu
linux: # cd cpu0/cpufreq/ondemand/
linux: # ls -1
ignore_nice_load
sampling_rate
sampling_rate_max
sampling_rate_min
up_threshold

linux: # cat sampling_rate_max
55000000
linux: # cat sampling_rate_min
55000
```

These times are measured in microseconds, denoting the minimum and maximum sampling rate. These values are read-only, and predetermined by the kernel as a function of P-state transition latency.

```
linux: # cat sampling_rate
110000
```

sampling_rate is a read-write file controlling how often the ondemand governor checks CPU utilization and tries to increase the CPU frequency at this rate. This field is in units of microseconds.

```
linux: # cat up_threshold
80
```

up_threshold is a read-write file showing the CPU-utilization threshold. Whenever the current utilization is more than up_threshold, the ondemand governor will increase the frequency to the maximum.

```
linux: # cat ignore_nice_load
1
```

ignore_nice_load is a read-write field that tells ondemand to treat time spent in /textitniced tasks as idle time.

4 ondemand governor optimizations

4.1 Changes between 2.6.9 and 2.6.16

Once the ondemand governor started getting used more widely, there was a lot of community feedback and patches to improve the algorithm. Several significant changes that went in since the original ondemand follow.

Automatic down-scaling of frequency The original ondemand algorithm, whenever it noticed a low utilization (less than 20% busy) reduced the frequency one-by-one through a range of values supported by hardware. This conservative approach was intended to minimize performance impact. But, as it started getting used more widely, we did not notice any performance issues due to the algorithm in general and there was opportunity to do more aggressive frequency reduction. Thanks to Eric Piel and his patch to this effect, the ondemand algorithm frequency down-scaling was changed to jump directly to the lowest frequency that can keep the CPU ~80% busy. This saves more power and enables the algorithm to go to right frequency in one hop under steady-state conditions.

Coordination of frequencies in software cpufreq supports multiple processors sharing the same frequency due to the hardware design. Say, in a particular implementation, different processor cores on a processor package are dependent on each other in terms of frequency. cpufreq supports it by managing these two CPUs together as one entity. In order to support this setup, the ondemand governor also has to manage the frequency of this entity based on the utilization of these two CPUs. The ondemand governor was changed to look at the utilization of all CPUs that are dependent this way and change the frequency of all of them based on highest utilization among the group.

4.2 Changes under investigation

There are a few other changes to the algorithm that are currently being investigated and can get into the base kernel in immediate future.

Unify up-scaling and down-scaling paths The original ondemand governor had a tunable to change the rate of ondemand CPU usage polling to increase the frequency and ondemand CPU usage polling, and an independent tunable to decrease the frequency. By default, the CPU usage polling to decrease the frequency was 10 times slower than the CPU usage polling to increase the frequency. The main reason for having this tunable was to keep any performance loss due to ondemand to a minimum. But over a period of ondemand usages, we have noticed that there is no advantage to having this tunable. In recent kernels, default sampling interval for frequency decrease is same as sampling rate for increasing the frequency.

By removing this option for different up- and down-scaling sampling frequency, we can cut the path length in ondemand sampling by half, which will be critical given how frequently we do the sampling.

Parallel calculation of utilization The original ondemand was doing the sampling and utilization in a centralized way for all CPUs. This does not scale well with increase in logical CPUs. One optimization is to have this sampling done at a per-cpu or per-domain having the shared frequency level instead of centralized sampling. Also, we can remove the locks/semaphore in the ondemand sampling path that can make ondemand scale well with increase in number of CPUs.

Dedicated workqueue ondemand has been using keventd and the generic workqueue interfaces to schedule the callbacks for periodic sampling. This callback would get called on one particular CPU, and ondemand sampling will run in context of keventd. One complication here, however, is if we want to change the frequency for a group of CPUs sharing the frequency, we may end up moving this particular process to a different CPU to make some calls to change the frequency. But we will be holding onto keventd from the original CPU and we may be delaying some other service that needs keventd. So, another change that adds value is to have dedicated kernel threads for ondemand and do the sampling and changing frequencies in the context of that particular kernel thread.

5 Coordination of P-states

With more than one logical package per physical socket, there are different kinds of frequency dependencies. This dependency is mainly due to hardware implementation and if OS knows about these dependencies, it can make more informed frequency decisions. There are different coordination schemes that can be implemented on any system.

There are four coordination schemes of interest. In the first two, the OS is ignorant of hardware dependencies. In the remaining two, the OS is aware of hardware dependencies.

5.1 Hardware coordination without OS knowledge

The hardware can do the coordination among these dependent logical CPUs internally without the knowledge of the OS. One way to implement it: hardware maintains multiple sets of registers to store the frequency requested by different logical processors, and then picks the maximum frequency requested by the group of these dependent CPUs to enforce that frequency on all CPUs belonging to the group. Hardware doing this coordination transparently will mean that OS still thinks each CPU is running on its own frequency.

This scheme has both an advantage and a disadvantage. The advantage is that no change is required in the OS to support this. cpufreq will still think that each CPU is independent and there will be different /sys/devices/system/cpu/cpuX/cpufreq directories for each CPU, even though they are dependent.

The disadvantage is that this can lead to bad decision making at times, as in the following example. CPU 0 and CPU 1 are dependent logical CPUs and can run at one constant frequency. Say at a given point in time, CPU 0 is at highest frequency (due to its load) and CPU 1 asks for a lower frequency. Hardware will do the coordination and run both CPUs at the higher frequency. But the OS will think CPU 1 is running at a lower frequency. On the next ondemand polling, CPU 1 will again notice that the CPU is idle (as it is actually still running at higher frequency than requested) and try to reduce the frequency further, even though the first lower request had no effect. Now if CPU 0 goes idle and lowers its frequency below CPU 1, then CPU 1 is now the maximum and it may run for a short time at a speed that is slower than it would have requested if it were an independent CPU.

5.2 BIOS coordination without OS knowledge

This scheme is very similar to Hardware coordination without OS knowledge. The only difference is that the BIOS does the actual coordination instead of hardware. BIOS can keep track of frequency requests from different CPUs in its own private space, pick the highest request and then make hardware calls to set the frequency at that highest request. The advantage and disadvantage is same as above, but with an additional disadvantage. Anything that runs in BIOS has to trap through SMM and this can result in an order of magnitude higher latency than the hardware coordination.

5.3 Hardware/BIOS coordination with OS knowledge

Similar to the two schemes above, except now the OS knows that this particular group of CPUs is dependent on each other. The OS will now know that hardware coordination is present and hardware can have additional interfaces, so that OS knows the frequency of a particular CPU over time. The OS can either manage each CPU independently (with a separate cpufreq directory for each CPU) or can do coordination in software and manage all the dependent CPUs as one unit (with one cpufreq directory for all the dependent CPUs).

5.4 Software coordination

In this scheme, the OS determines the logical CPUs that are dependent and does all the coordination required in software. The OS can monitor all the dependent CPUs together and make one frequency change request to hardware, depending on information from all the dependent CPUs. In this case Linux will have one cpufreq directory for all the dependent CPUs and /sys/devices/system/cpu/cpuX/cpufreq, for all X in dependent CPUs, will be a symbolic link to one common cpufreq interface. /sys/devices/system/cpu/cpuX/cpufreq/affected_cpus interface will provide the list of CPUs that share same frequency, in case of software coordination. Also, note that in this case, the OS may depend on the BIOS to know what particular logical CPUs are dependent on each other. ACPI 3.0 provides the

_PSD interface where the OS can get this information about all the CPUs that are dependent in terms of frequency.

5.5 Linux support for coordination

Linux-2.6.17-rc*-mm* has support for the ACPI 3.0 _PSD method, and the speedstep-centrino and acpi-cpufreq drivers can make use of this interface to determine the dependent CPUs (and also the mode of coordination—Hardware or Software coordination). Both the cpufreq and ondemand governor have supported software coordination since Linux-2.6.14. So, Linux can run the two OS-aware coordinations schemes if the BIOS exports the specific ACPI interfaces.

Currently, most of the BIOSes do not provide any coordination information to OS and Linux will run in the hardware or BIOS coordination schemes.

6 Performance Measurements

6.1 Methodology used for measurements

In order to be able to compare different improvements to cpufreq algorithms, we have set up an experimental system, running a standard web server workload over SSL. We have measured 12V DC current to server CPUs. Loading clients were connected to server by direct 1Gb links. We choose to not use HTTP accelerators such as TUX, because our primary goal was not getting record scores, but to have a well-defined dynamic server load. Pairs of our graphs show consumed power and number of conforming client-server connections vs. number of requested connections. The tested system is a 4-socket Xeon MP dual-core and hyper-threaded machine (with a total of 16 logical processors) with 8GB of RAM.

6.2 Experiment setup

Power consumed by the system can be measured by special power meters such as *WattsUp?* or manually by means of various sensors, introduced into the system under test. In the latter case, one measures separately voltage (U) (or treats it as a known value) and current (I), running through the system and then multiplies acquired values to get consumed power ($P = U * I$). In the case of internal DC supply voltages ($12V$), one can sample current with 100Hz frequency and get pretty high accuracy. We used one of the cheapest USB DACs around—PMD-1208LS, other choices being devices from LabJack or even soundcard input. In order to measure current with the DAC one needs to convert it into voltage. This could be done by inserting a milliOhm-range resistor into a powering wire, or measure a magnetic flux around the wire with Hall-effect sensor. We used a second approach with split-core sensors CR5410S from CRMagnetics which could be wrapped around the wire without a need to break it. This setup allowed for about 1-Watt variation in power measurements from run to run, which is more than adequate, considering more than 600 Watt peak power consumption. Thus we choose to show on graphs results as is, without additional averaging.

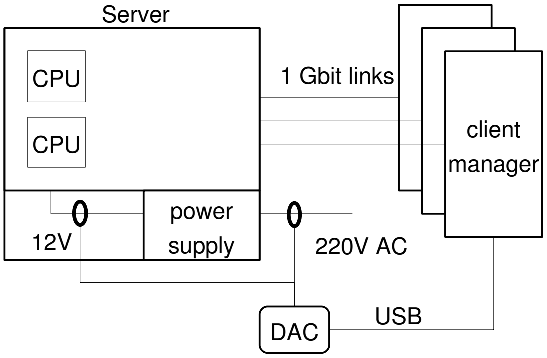

Figure 3: Experiment setup

6.3 No power management, userspace, original ondemand

The following picture shows the results on a 16-logical-CPU system with different governors. performance and powersave governors have power delta of about 10%, while ondemand and userspace stay in between. Performance degradation is significant with powersave and barely visible with dynamic governors.

6.4 Original ondemand and experimental governors

These graphs represent new experiments with the ondemand:

2.6.9 First ondemand, from kernel 2.6.9, appears to not save any power in such a system, while trashing performance. Included here for reference.

clean Removed duplicating down-sampling calculations.

parallel Introduction of own workqueue and scheduling of utilization calculations on each CPU group.

fastcheck Make a check of the utilization somewhat faster in the case of setting high frequency.

idle Use idle_notifier to find exact idle times.

7 Future Work

7.1 Impact on other subsystems—Scheduler changes

Today the Linux scheduler does not have any knowledge of frequency at which a processor is running. It assumes each CPU on an SMP system has the same amount of horse-power and tries to balance the load equally across CPUs. Recent smpnice patches have added the process priority information into the scheduler. We also need to add the CPU horsepower into the scheduler as well.

[5] talks about making the scheduler aware of frequency dependencies across logical processors in a multi-core environment. The scheduler can change the load balancing behavior, depending on whether a system wants to optimize performance or power. For instance, on a DP[1] system with each package being dual-core, and with performance policy, two active threads should run on different packages, keeping one core on each package idle. This will optimize resource utilization of all the shared resources across two cores on a package. When the same setup is running in power optimized policy, two active threads will run on two cores of a single package, allowing other two cores of other package to go idle and also to lower frequency/voltage.

Even in normal SMP (without threads or cores), to get maximum power savings, the scheduler should try to get one CPU 100 percent busy, even though with some loss in response time, before moving tasks to next CPU. This is a yet-to-be-explored area at this time.

7.2 Callback and micro-accounting for idle

The current ondemand governor depends on the idle/busy statistics collected at the scheduler ticks. If at the tick instance the CPU was idle, then whole tick is considered idle and vice-versa. But, if we can do a micro-accounting of idle time then we get a more accurate number of time spent idle and time

[1] DP = Dual Package.

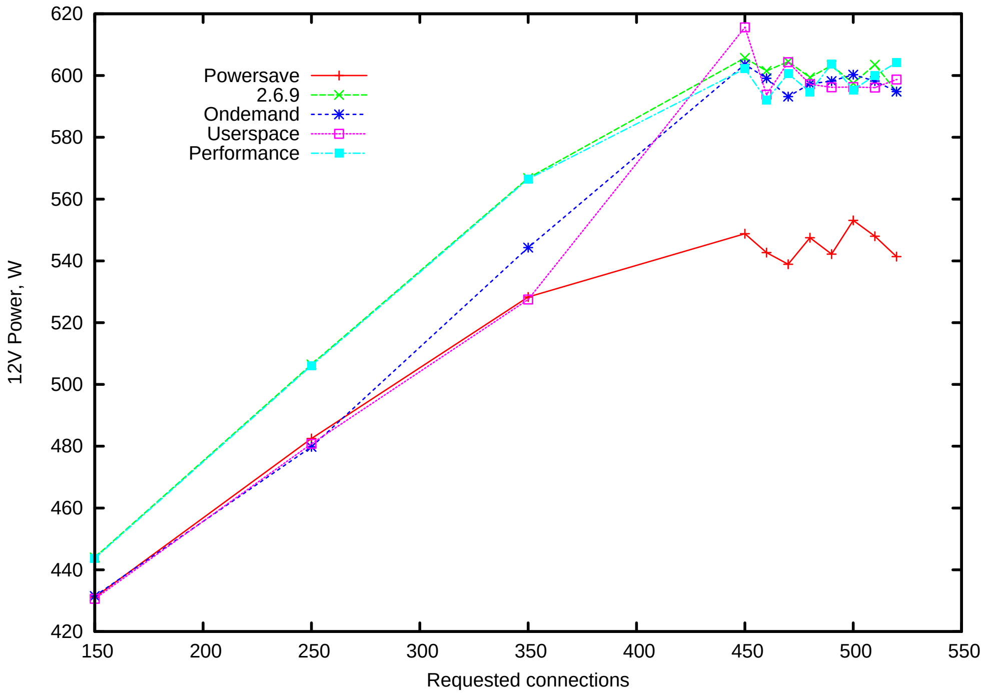

Figure 4: Power consumption with original governors

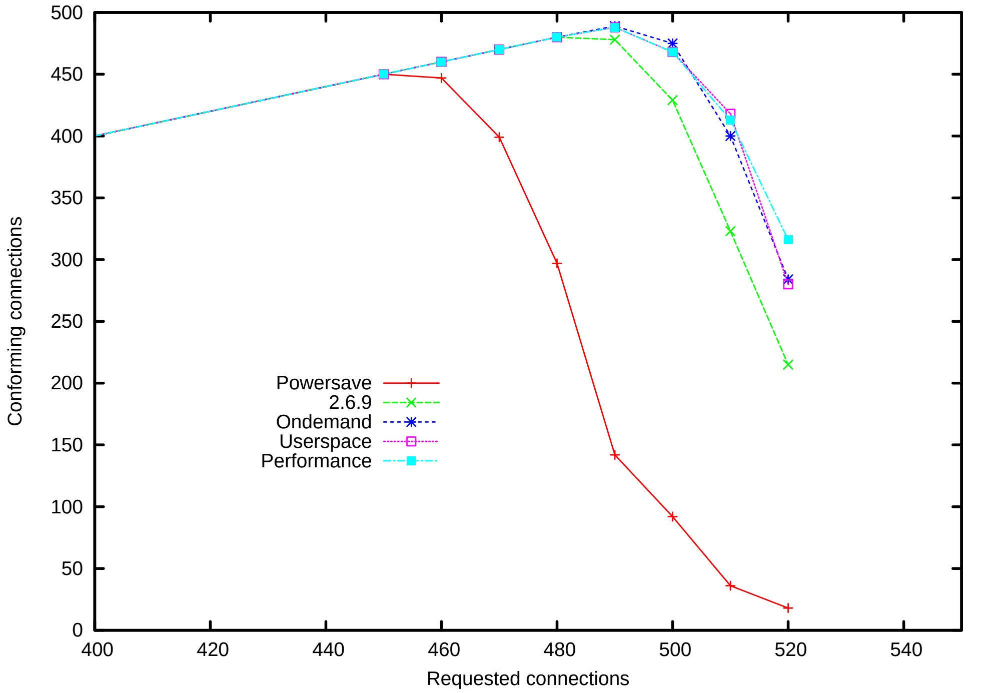

Figure 5: Performance with original governors

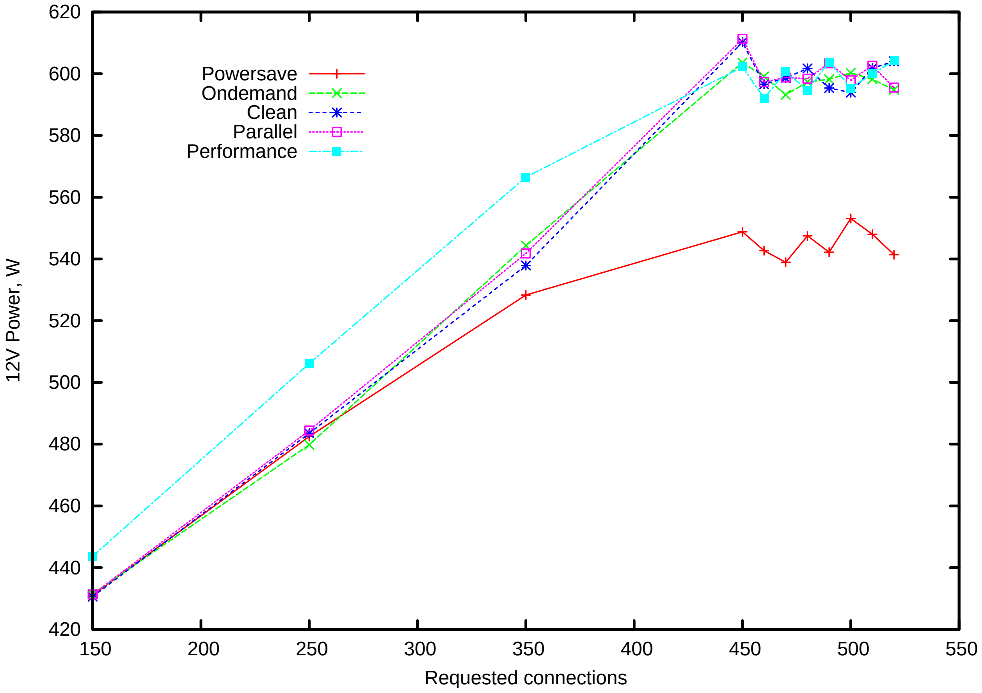

Figure 6: Power consumption with experimental governors

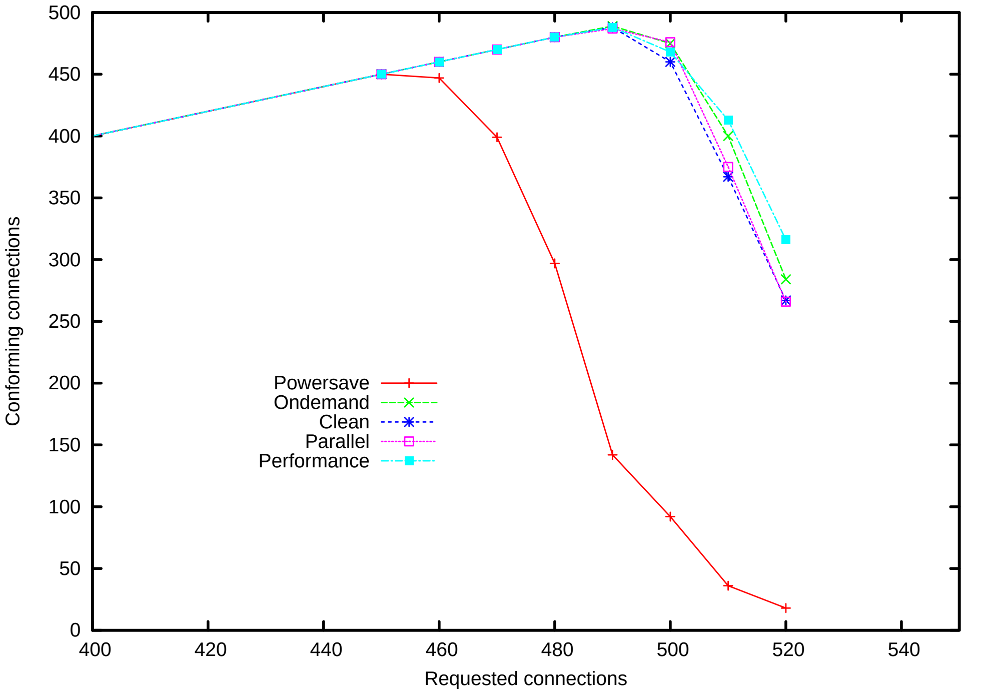

Figure 7: Performance with experimental governors

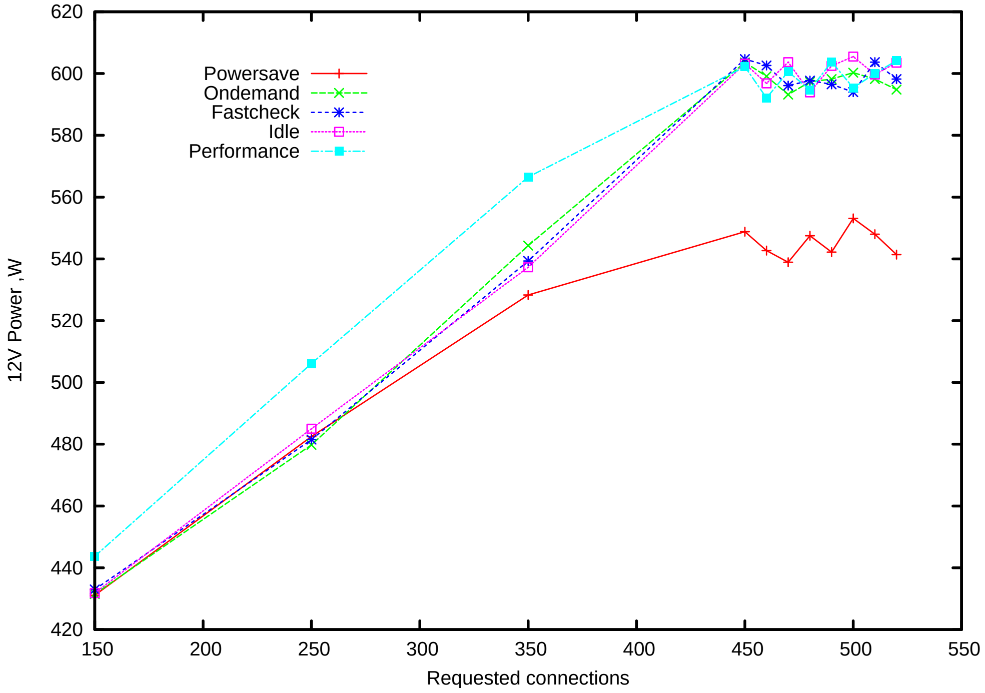

Figure 8: Power consumption with experimental governors

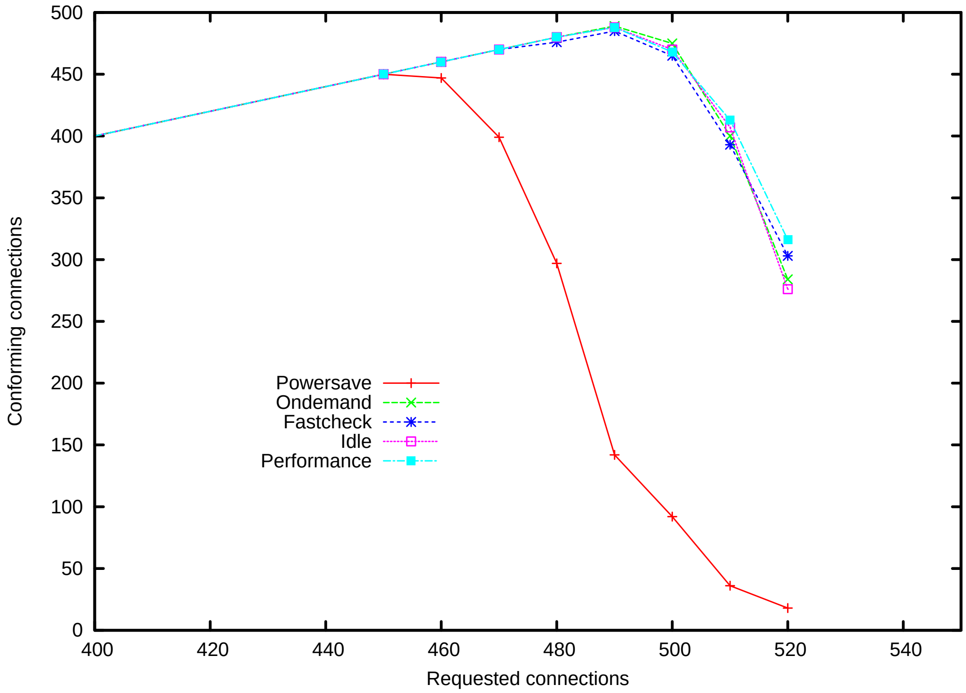

Figure 9: Performance with experimental governors

spent doing useful work. The kernel can do the micro-accounting by noting the time of entry and exit of idle routine and interrupts.

If the idle handler across architecture has to do accounting for exact idle time, ondemand will have more accurate idle/busy data and can take better frequency decisions. Though doing micro-accounting for all the states like user, kernel, nice, etc. may have some overhead, doing it only for idle/non-idle should be relatively easy. Andy Kleen has implemented idle notifier callbacks for X86_64, and our experimental governor makes use of this infrastructure. We needed to keep overhead of such accounting low, because it is called during each interrupt enter/exit.

7.3 Real time threads and impact

The ondemand governor runs in the context of a kernel thread and the real time processes running on the system may get higher priority and run before the ondemand governor gets a chance to increase the frequency. This is the current issue with ondemand and real time threads. There is no clean solution for this problem, as if we try to increase the frequency before the real-time process starts, the transition latency to increase the frequency will delay the start of the real-time process and also, the real-time process may not run for a long time, negating the whole purpose of increasing the frequency. One solution to this is to have some callbacks from the scheduler, before it schedules the real-time threads, to ondemand governor, which can then increase the frequency giving the benefit of increased frequency to real time threads. Note that this has to be a special case only for real-time threads, as adding some additional checks/callbacks like this for normal threads in context switch path will be a problem as it is a common case and should be be delayed. More ideas on how to solve this issue, as well as patches to solve this problem, are welcome :-).

8 Acknowledgments

Thanks to our colleagues at Intel Open Source Technology Center for their continuous support. Thanks to efforts of many developers and testers in open source community. Special thanks to Len Brown, Dominik Brodowski, Andi Kleen, Eric Piel, and Thomas Renninger for all the support, feedback, and patches.

References

[1] cpufreq-stats documentation. Documentation/cpu-freq/cpufreq-stats.txt in Linux kernel source.

[2] cpufrequtils project page. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html.

[3] Dominik Brodowski. Current trend in linux kernel power management, linuxtag 2005. http://www.free-it.de/archiv/talks_2005/paper-11017/paper-11017.pdf.

[4] Len Brown et al. Acpi in linux, ols 2005. http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf.

[5] Suresh B. Siddha et al. Chip multi processing aware linux kernel scheduler, ols 2005. http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf.

[6] Venkatesh Pallipadi. Enhanced intel speedstep technology and demand-based switching on linux, intel software net. http://www.intel.com/cd/ids/developer/asmo-na/eng/195910.htm.

[7] Linus Torvalds. Linus about kernel governor on lkml. http://marc.theaimsgroup.com/?l=linux-kernel&m=103056055008566&w=2.

Disclaimer

The opinions expressed in this paper are those of the authors and do not necessarily represent the position of the Intel Corporation.

Linux is a registered trademark of Linus Torvalds.

Intel is a registered trademark of Intel Corporation.

All other trademarks mentioned herein are the property of their respective owners.